Information and Computation 232 (2013) 10–18

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Compressed property suffix trees ☆

Wing-Kai Hon [a], Manish Patil [b], Rahul Shah [b,*], Sharma V. Thankachan [b]

[a] Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
[b] Department of Computer Science, Louisiana State University, Baton Rouge, LA, USA

ARTICLE INFO

Article history:
Received 2 December 2011
Available online 18 September 2013

Keywords:
Property matching
Suffix trees
Property suffix trees

ABSTRACT

Property matching is a biologically motivated problem where the task is to find those occurrences of an online pattern P in a string text T (of size n), such that the matched text part satisfies some conceptual property. The property of a string is a set π of (possibly overlapping) intervals $\{(s_1, f_1), (s_2, f_2), \ldots\}$ corresponding to the part of text and an occurrence of a pattern $P = T[i, \ldots, (i + |P| - 1)]$ is a valid output only if $T[i, \ldots, (i + |P| - 1)]$ is completely contained in at least one interval $(s_j, f_j) \in \pi$. The indexing version of this problem was introduced by A. Amir (2008), where the text is preprocessed in $O(n \log \sigma + n \log\log n)$ time and an $O(n \log n)$ bits index, named Property Suffix Tree (PST) is maintained. PST can perform property matching in $O(|P| \log \sigma + occ_\pi)$ time, where occ_π is the number of occurrences of P in T satisfying the property. T. Kopelowitz (2010) considered the dynamic version of this problem where intervals can be added or deleted. However, all these indexes take space linear to the size of text ($O(n \log n)$ bits), which can be much more than the size of the text ($n \log \sigma$ bits). In this paper, we propose the first index for property matching occupying space close to the entropy compressed space requirement of the text. Our compressed index takes $|CSA| + n(2 + \epsilon + o(1))$ bits space and performs query answering in $O(t(|P|) + \frac{1}{\epsilon}(1 + occ_\pi) t_{SA})$ time, where $|CSA|$ is the size of compressed suffix array of T, $t(|P|)$ be the time for searching a pattern of length $|P|$ in CSA, t_{SA} is the time for computing the suffix array value and $\epsilon > 0$ is a constant. We also introduce a dynamic index, which takes $|CSA| + O(n + |\pi| \log n)$ bits space and performs query answering in $O(t(|P|) + (1 + occ_\pi) \log n (t_{SA} + \log n / \log\log n))$ time and can update (insert/delete) an interval (s, f) in $O((f - s)(\log n + t_{SA}))$ time.

© 2013 Elsevier Inc. All rights reserved.

1. Introduction

Given a text T of size n over an alphabet set Σ of size σ, the fundamental problem in text indexing is to preprocess this text and maintain an index such that whenever an online pattern P comes as query, all the occurrences of P in T can be reported efficiently. A classic data structure for solving this problem is suffix tree which can perform pattern matching in optimal $O(|P| + occ)$ time, where occ is the number of occurrences of P in T. Another classical data structure is suffix array with a query time of $O(|P| \log n + occ)$ [1]. This can be improved to $O(|P| + \log n + occ)$ using an additional data structure called LCP array [1]. However, these indexes take $O(n \log n)$ bits space, which can be much more than the optimal $n \log \sigma$ bits. For example in genome data ($\Sigma = \{A, G, C, T\}$), $\log \sigma$ is 2, where as $\log n$ is around 30. As the memory of computers is limited, this gives a clear motivation to have compressed data structures for handling large data. This long standing problem

☆ A preliminary version appears in *Proceedings of Data Compression Conference*, 2011. This work is supported in part by Taiwan NSC Grant 99-2221-E-007-123-MY3 (Wing-Kai Hon), and by US NSF Grant CCF-1017623, CCF-1218904 (Rahul Shah).

* Corresponding author at: 3122-A Patrick F. Taylor Hall, LSU School of Electrical Engineering and Computer Science, Baton Rouge, LA 70803, USA. Fax: +1 225 578 1465.

E-mail addresses: wkhon@cs.nthu.edu.tw (W.-K. Hon), mpatil@csc.lsu.edu (M. Patil), rahul@csc.lsu.edu (R. Shah), thanks@csc.lsu.edu (S.V. Thankachan).

0890-5401/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ic.2013.09.001

was positively answered by compressed suffix array proposed by Grossi and Vitter [2] and FM-index proposed by Ferragina and Manzini [3]. Different versions of these indexes are available which achieve different space–time trade-offs (see [4] for an excellent survey). Both these indexes can efficiently handle general pattern matching in compressed space. Focus of this paper is on a special kind of pattern matching called property matching.

The property of a text is a set π of (possibly overlapping) intervals $\{(s_1, f_1), (s_2, f_2), \ldots\}$, such that $T[s_j, \ldots, f_j]$ satisfies some conceptual property. Property matching is a variant of classic string matching with an additional constraint that an occurrence of a pattern $P = T[i, \ldots, (i + |P| - 1)]$ is completely contained in at least one interval $(s_j, f_j) \in \pi$. The main motivation of this problem comes from biological applications. In molecular biology, it has long been a practice to consider special genome areas by their structures [5]. For example, the following problem can be modeled as property matching: find all the occurrences of a give pattern in a genome, provided it appears in a repetitive genomic structure such as tandem repeats, SINEs (short Interspersed Nuclear Sequences), or LINEs (line Interspersed Nuclear Sequences) [6].

A new pattern matching paradigm has recently attracted a lot attention where the given text T is weighted [7–9]. At each position in a weighted text, a set of characters appears instead of a fixed single character occurring in a normal string. Such a set of characters appearing at a particular text position also includes probability of appearance of each of it's member character at that position. Weighted text is common in various applications of computational biology. The problem of "motif discovery" is one among these applications that can be benefited due to improvements in the area of property matching. It is essentially a problem of exact pattern matching on weighted text and is also known as "weighted matching". In this problem, task is to seek whether or not, and where a given motif (pattern) occurs in a weighted text. Amir et al. [7] present a reduction of this problem to property matching and show how off-the-shelf techniques for property matching can be used to solve it. The proposed reduction also yields solutions to other problems common in pattern matching community such as, scaled matching, swapped matching, and parameterized matching in weighted text [7].

It is easy to solve algorithmic version of property matching problem in time linear to the size of text. Any standard string searching algorithm can be used to retrieve all the occurrences and later filter out those occurrences which are within the intervals in π. When it comes to indexing problem, our task is to answer the query in time proportional to the size of pattern (not to the size of text) and the number of outputs. Normal string searching data structures like suffix trees or suffix arrays cannot be directly applied in this case as they report all the occurrences (occ) of P in T. Note that occ can be much more than occ_π, where occ_π is the number of occurrences of P satisfying the text property π. Hence the above strategy is not optimal.

Most of the indexing solutions augment suffix tree/array data structure with extra information, such that a query time proportional to $|P|$ and occ_π can be obtained. However, all the previously known indexes take $O(n \log n)$ bits and are not space efficient. In this paper, our objective is to design compressed space indexes for property matching problem. The suffix tree/array data structure can be replaced by compressed suffix tree/array and can be used as a black box. The challenging part is to come up with an encoding scheme for compressing the augmented information while maintaining efficient query capabilities, and is the key contribution of this paper.

1.1. Previous results

Property matching problem was first studied by Amir et al. [7]. They introduced a new data structure called property suffix tree (PST), which can solve the problem in $O(|P| \log \sigma + occ_\pi)$ time. Their structure is basically a suffix tree augmented with extra information so as to report only occ_π outputs. PST takes $O(n \log n)$ bits space and can be constructed in $O(n \log \sigma + n \log \log n)$ time. Later Iliopoulos and Rahman [10] proposed an alternative index called IDS-PIP, based on Range Maximum Queries (RMQ), which can be constructed in linear time. However their index could not correctly handle the case when the intervals are not disjoint. Juan et al. [11] modified IDS-PIP index for handling the general case, which takes $O(n \log n)$ bits space and can answer the query in optimal $O(|P| + occ_\pi)$ time. Another direction of research in this topic is to consider the dynamic version of the problem, where new intervals can be inserted or existing intervals can be deleted. Kopelowitz [12] proposed an index where insertion or deletion of an interval (s, f) can be performed in $O(f - s + \log \log n)$ time while maintaining linear index space and optimal query time. Though the open problem of designing a compressed index for property matching remains unanswered, Zhao and Lu [13] (parallelly and independently to our work) have proposed a space efficient index for property matching, which is truly succinct only when $\pi = O(n/\log n)$.

1.2. Our results

In this paper, we propose a space efficient index for property matching, namely Compressed Property Suffix Tree (C-PST). Theorem 1 summarizes the result for C-PST, when intervals in the text property π are fixed. Theorem 2 summarizes the result for a dynamic C-PST, where text property π is dynamic i.e. new intervals can be added to or existing intervals can be deleted from π.

Theorem 1. *Given a text T of length n and a property π, a C-PST can be maintained in $|CSA| + n(2 + \epsilon + o(1))$ bits such that property matching queries can be answered in $O(t(|P|) + \frac{1}{\epsilon}(1 + occ_\pi)t_{SA})$ time, where P is the online query pattern, $|CSA|$ is the size of compressed suffix array of T, $t(|P|)$ is the time for searching P in CSA, t_{SA} is the time for computing the suffix array (SA) value and $\epsilon > 0$ is a constant.*

Fig. 1. Example of property matching for pattern $P =$ "CAT" over text T with property $\pi = \{(3,7),(10,13),(18,27),(22,30),(37,38)\}$.

Theorem 2. *Given a text T of length n and a property π, a dynamic C-PST can be maintained in $|CSA| + O(n + |\pi|\log n)$ bits space such that property matching queries can be answered in $O(t(|P|) + (1 + occ_\pi)\log n(t_{SA} + \log n/\log\log n))$ time and update (insert or delete) of an interval (s, f) can be performed in $O((f - s)(\log n + t_{SA}))$ time.*

1.3. Paper organization

The remainder of this paper is organized as follows: In Section 2, we first formally define notion of property and property matching and then introduce basic tools which form the building blocks of our indexes. In Section 3, we explain our index i.e. Compressed Property Suffix Tree (C-PST) for the case when property of a text (π) is static. We dedicate Section 4 for a dynamic version of C-PST, which can handle dynamic text property i.e. intervals can be added to or deleted from π. Finally, we draw conclusions in Section 5.

2. Preliminaries and definitions

Let T be the given string text of length n, which is to be indexed and P be the online query pattern of length $|P|$. All characters in T and P are drawn from the same alphabet set Σ of size σ. A substring of T which starts at location i and ends at location j is denoted by $T[i,\ldots,j]$. The notion of property and property matching are formally defined as follows.

Definition 1. (See [7].) A property π of a string T of length n is a set of intervals $\pi = \{(s_1, f_1), (s_2, f_2), \ldots\}$ where for each $1 \leqslant i \leqslant |\pi|$ it holds that: (1) $s_i, f_i \in 1,\ldots,n$, and (2) $s_i \leqslant f_i$. The size of property π, denoted by $|\pi|$ is the number of intervals in the property.

Definition 2. (See [7].) Given a text T with property π and pattern P, we say that P matches $T[i,\ldots,j]$ under property π if $P = T[i,\ldots,j]$ and there exists $(s_k, f_k) \in \pi$ such that $s_k \leqslant i \leqslant j \leqslant f_k$.

Fig. 1 shows an example of a sample text T with property $\pi = \{(3,7), (10,13), (18,27), (22,30), (37,38)\}$ and pattern $P =$ "CAT". Though the given query pattern P occurs in text T at 5 distinct locations i.e. $occ = 5$, there are only 2 occurrences of P ($T[4,\ldots,6]$, $T[22,\ldots,24]$) which satisfy the text property π i.e. $occ_\pi = 2$.

2.1. Suffix trees and suffix arrays

Suffix trees [14] and suffix arrays [1] are two classic data structures for online pattern matching queries. For a text $T[1\ldots n]$, substring $T[i\ldots n]$, with $i \in [1,n]$, is called a suffix of T. The suffix tree for T is a lexicographic arrangement of all these n suffixes in a compact trie structure, where the ith leftmost leaf represents the ith lexicographically smallest suffix. Whereas suffix array $SA[1\ldots n]$ is an array of length n, such that $SA[i]$ is the starting position of the ith lexicographically smallest suffix of T. Suffix array has an important property that the starting positions of all suffixes with the same prefix are always stored in a contiguous region in SA. Based on this property, the suffix range of a pattern P in SA is defined as the maximal range $[\ell, r]$ such that for all $j \in [\ell, r]$, $SA[j]$ is the starting point of a suffix of T with P as a prefix. Both suffix trees and suffix arrays take $(n\log n)$ bits space and can perform pattern matching in $O(|P| + occ)$ and $O(|P|\log n + occ)$ time respectively. The query time of suffix array data structure can be improved to $O(|P| + \log n + occ)$ using an additional $O(n\log n)$ bits data structure called LCP array [1].

Space efficient versions of suffix trees and suffix arrays, known as compressed suffix trees (CST) [15] and compressed suffix arrays (CSA) [2] take space close to the size of text. In this paper we use CSA only as a black box. Hence we represent the size of CSA by $|CSA|$, the time for searching a pattern (or finding the suffix range) of length $|P|$ by $t(|P|)$ and the time for accessing a suffix array value as t_{SA}. Different versions of these indexes are available which achieve different space–time trade-offs (see [4] for an excellent survey). For example, the latest version by Belazzougui and Navarro [16] takes $nH_k + o(nH_k) + O(n + n\log n/s)$ bits space such that $t(|P|) = O(|P|)$ and $t_{SA} = s$, where H_k is the kth order empirical entropy of T and $s = \omega(\log_\sigma n)$ is a sampling step.

2.2. Range maximum query structure on a static array

Let A be an array of length n, a Range Maximum Query (RMQ) asks for the position of the maximum between two specified array indices $[i, j]$. i.e. the RMQ should return an index k such that $i \leqslant k \leqslant j$ and $A[k] \geqslant A[x]$ for all $i \leqslant x \leqslant j$. Although solving RMQ is as old as Chazelle's original paper on range searching [17], many simplifications and improvements have been made [18], culminating in Fischer and Heun's $2n + o(n)$ bit data structure [19]. All these schemes can answer RMQ

in $O(1)$ time when array A is static. For our purpose, we shall use the following results by Fischer and Heun [19]: When input array A is static, RMQ can be answered without accessing A in $O(1)$ time, by maintaining an additional structure of size $2n + o(n)$ bits. If we are allowed to access the array A during query time, this additional structure can be maintained in $\frac{n}{c(n)}(2 + o(1))$ bits space with query time $O(c(n))$, where $c(n)$ can be any positive integer function.

Lemma 1. *Let A be an array of size n and t_{access} be the time for accessing an element in A, then the Range Maximum Queries on A can be answered in $O(\frac{1}{\epsilon} \times t_{access})$ time by maintaining an index of size $\epsilon n(1 + o(1))$ bits.*

Proof. Choose $c(n) = 2/\epsilon$, where $\epsilon > 0$ and we assume that the array A is stored separately and can access the element $A[i]$ for any given $i \in [1, n]$ in t_{access} time (if the array A is not stored explicitly). □

2.3. Range maximum query structure on a dynamic array

In this section, we show how to maintain an RMQ structure on A, when the values of elements in A can be updated (we are not considering the case of insert or delete of elements, hence the length of array remains the same). The traditional RMQ techniques based on cartesian trees may not work in this case. But this problem can be handled by a balanced binary tree of n leaves as follows, the leaves are numbered from 1 to n from left to right. Each leaf i stores the value $A[i]$. Each internal node stores the index of left most and right most leaf in its subtree. It also stores the index of a leaf in its subtree with the maximum value. Now any given range can be split into at most $2 \log n$ subranges such that, each of this subrange is exactly the subtree of an internal node. Since maximum among each subrange is stored in the corresponding internal node, the overall maximum can be computed by comparing $2 \log n$ elements in $O(\log n)$ time. Whenever the value of a leaf changes, we need to update the values on all internal nodes which are in the path from that leaf to root. Since the height of the tree is $O(\log n)$, this operation can be bounded by $O(\log n)$ time.

Lemma 2. *Let A be a dynamic array of size n, such that the values in A can be updated and t_{access} be the time for accessing an element in A, then a balanced binary tree structure of $O(n \log n)$ bits can be maintained to answer Range Maximum Queries and updates in $O(\log n)$ time.*

Proof. Follows from the discussion above. □

Lemma 3. *Let A be a dynamic array of size n, such that the values in A can be updated and t_{access} be the time for accessing an element in A, then a balanced binary tree structure of $O(n)$ bits can be maintained to answer Range Maximum Queries in $O(t_{access} \log n)$ time and updates in $O(\log n)$ time.*

Proof. Divide array A into $(n/\log n)$ contiguous blocks of size $\log n$ and the maximum value in each block is stored in another array A' of length $(n/\log n)$. A dynamic RMQ structure can now be maintained on A' which takes only $O(n/\log n \times \log n) = O(n)$ bits (from Lemma 2). When the query range is exactly between two blocking boundaries of A, this sampled RMQ structure can return the block in which the maximum exists in $O(\log n)$ time. Then the exact maximum can be computed by checking all $\log n$ elements in that block in $O(\log n)$ time. When the range is not exactly between two blocking boundaries, we can find the maximal range $[i', j']$ which is exactly between two blocking boundaries. The RMQ on this new range can be performed in $O(\log n)$ time and the RMQ on two small subranges (which are partially within a block) on either side of this maximal range can be performed by checking all elements within it. The highest value among the maximums obtained from these 3 subranges is the final RMQ answer. Note that we perform at most $3 \log n$ comparisons in the original array. Thus the query time is bounded by $O(\log n)$. Now when there is an update, we check if the new value is greater than the greatest value in the corresponding block. If so, we update A' and the corresponding balanced binary search tree also in $O(\log n)$ time. This completes the proof of Lemma 3. □

2.4. Bit vectors with rank/select

Let B be a bit vector of length n, the *rank* and *select* operations are defined as, $rank(k) = \sum_{i=1}^{k} B[i]$ and $select(k) = i$, where $A[i] = 1$ and $rank(i) = k$.

Several strategies have been developed to efficiently compute *rank* and *select* for bit vectors till date. Jacobson [20] was the first to provide a data structure of size $o(n)$ supporting *rank* operation in constant time. Though he also studied *select* operation, the solution was not optimal. Later, Clark and Munro [21] obtained constant time *rank* as well as *select*, using $o(n)$ extra space. These $n + o(n)$ solutions are asymptotically optimal for incompressible bit vectors. By exploiting the compressibility of bit vectors, constant time rank and select implementations with smaller bit vector representations have been obtained in [22] and [23]. In this paper, we use the following result by Raman et al. [23].

Table 1
Computing $end(i)$ and $length(i)$ for text T with property π (Fig. 1).

i	1	2	3	4	…	12	…	21	22	23	…	31	…	35	…
$end(i)$	0	0	7	7	…	13	…	27	30	30	…	30	…	30	…
$length(i)$	0	−1	5	4	…	2	…	7	9	8	…	0	…	−4	…

Lemma 4. *Given a bit vector B of length n, there exists a static data structure that can support both rank and select operations in constant time, by using $nH_0(B) + o(n)$ bits space and taking only $O(n)$ time to construct, where $H_0(B) \leqslant 1$ is the 0th order empirical entropy of B.*

In case of dynamic bit vectors, we only need a limited range of dynamic operations i.e. "flip" operations only ($flip(i)$ operation flips the bit in the ith position). There have been some results on dynamic succinct bit vectors [24–27] with the goal of supporting efficient *rank* and *select* operations while allowing "flip" operations. In this paper, we use the following result by Raman et al. [24].

Lemma 5. *Given a bit vector B of length n, an additional $o(n)$ bits index can be maintained to support rank, select and flip operations in $O(\log n / \log\log n)$ time.*

3. Compressed property suffix trees

In this section, we introduce the compressed index (C-PST) for property matching. C-PST consists of a compressed suffix array (CSA), a bit vector B (along with *rank–select* structures) and a Range Maximum Query (RMQ) structure. Since there are different versions of compressed suffix arrays available with different space–time trade-off's, we denote the size of CSA by $|CSA|$ and the time taken for retrieving the suffix array value $SA[i]$ by t_{SA}. The outline of query answering algorithm is as follows, first we perform the pattern matching in $O(t(|P|))$ time using CSA and obtain the suffix range $[\ell, r]$. Since we are not interested in all the values in suffix range, we use additional structures to retrieve only occ_π valid outputs. Before going to the details, we review the following notions of extents and maximal extent from the previous papers.

Definition 3. (See [12].) Given a text T with property π for every text location $1 \leqslant i \leqslant n$ and interval $(s, f) \in \pi$ such that $s \leqslant i \leqslant f$, the f is called an extent of i. The maximal extent of i is the extent of i of large value, or in other words, finish of the interval containing i which is the most distant from i. The maximal extent of i is denoted by $end(i)$. If for some location i there is no interval in π containing it, $end(i)$ is defined as NIL.

The following lemma explains the significance of maximal extents in property matching problem.

Lemma 6. *(See [12].) Given a text T with property π, a pattern P matches $T[i, \ldots, j]$ under property π if and only if $P = T[i, \ldots, j]$ and $j \leqslant end(i)$.*

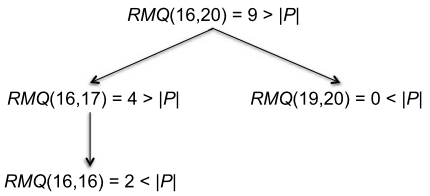
In this paper, we change the definition of $end(i)$ slightly as follows. Note that this definition will change only entries with $end(i) = \text{NIL}$, hence Lemma 6 is also valid according to the following new definition.

Definition 4. $end(i) = \max f_k$, such that $s_k \leqslant i$. Conceptually this represents the ending position (within the text) of the longest possible substring of T, whose starting position is i and is completely within an interval in π.

We define a function *length*, where $length(i) = end(i) + 1 - i$ is the length of maximal prefix of the suffix starting at location i in T and is completely within an interval in π (see Table 1 for an example). Now a pattern P matches at location i under property π if and only if the match starts at location i in T and ends on or before $end(i)$, i.e. $P = T[i, \ldots, (i + |P| - 1)]$ and $|P| \leqslant length(i)$. Table 1 shows computations of $end(i)$ and $length(i)$ based on the sample text T with property π depicted in Fig. 1. As it can be seen from Table 1, out of the 5 occurrences of pattern $P =$ "CAT" in the text T, only 2 occurrences (with starting locations $i = 4$ and 22) are completely contained in an interval $(s_j, f_j) \in \pi$ and also satisfy the condition $|P| \leqslant length(i)$. An important observation is that, $end(i) \leqslant n$ and is a non-decreasing function. This can be easily converted to another strictly increasing function $\zeta(i)$ as $\zeta(i) = i + end(i)$ i.e. $\zeta(i) < \zeta(i + 1) \leqslant 2n$. Therefore the function $\zeta(i)$ can be encoded as a bit vector $B[1 \ldots 2n]$, where $B[\zeta(i)] = 1$, else 0. Further we maintain *rank* and *select* structure over array B. Using these operations, the following functions can be computed in $O(1)$ time as follows

Table 2
Query answering for pattern $P =$ "CAT" on text T with property π (Fig. 1).

		Suffix Range for pattern $P =$ "CAT", $\|P\| = 3$					
i	...	16	17	18	19	20	...
$SA[i]$	...	12	4	22	31	35	...
$A[i] = length(SA[i])$	...	2	4	9	0	−4	...



$$\zeta(i) = select_B(i),$$

$$end(i) = select_B(i) - i,$$

$$length(i) = select_B(i) + 1 - 2i.$$

We define an array A, such that $A[i] = length(SA[i])$, which stores the *length* of text locations sorted in the lexicographic order of the suffix starting at that location. Note that $length(i)$ for $i = 1, \ldots, n$ can be stored in $2n + o(n)$ bits as it is an increasing function, but storing $A[i]$ directly is costly. However we only need to maintain RMQ structure on $A[i]$ in $\epsilon n(1 + o(1))$ bits (using Lemma 1) which can directly return the suffix with maximum *length*, within a given suffix range.

Lemma 7. *C-PST occupies* $|CSA| + n(2 + \epsilon + o(1))$ *bits space.*

Proof. Follows from the discussion above. □

3.1. Query answering

Here we show how to answer the property matching query using *CSA*, B array (along with *rank–select* structures) and RMQ over array A (we do not store array A explicitly).

Lemma 8. *Property matching can be performed in* $O(t(|P|) + \frac{1}{\epsilon}(1 + occ_\pi)t_{SA})$ *time using C-PST.*

Proof. First we match the pattern P using CSA in $O(t(|P|))$ time and obtain the suffix range $[\ell, r]$. Then by initiating a range maximum query on array A with ranges as $[\ell, r]$, we can obtain an index k, such that $\ell \leqslant k \leqslant r$ and $SA[k]$ be the suffix with the maximal prefix within an interval in π. In order to check if it is a valid output, we check if $length(SA[k]) \geqslant |P|$. If so, we report the position $SA[k]$ as a valid output and we continue this procedure recursively in the subranges $[\ell, k-1]$ and $[k+1, r]$. Whenever this condition is violated, we will stop recursing that subrange further. Table 2 shows these steps for answering a query with pattern $P =$ "CAT" on the text T with property π depicted in Fig. 1. To retrieve all occ_π outputs, we may do this procedure at most $(2\,occ_\pi + 1)$ times, hence the total time for reporting can be bounded by $O((1 + occ_\pi)t_{SA})$. □

Lemmas 7 and 8 collectively prove Theorem 1. Out of different versions of compressed suffix arrays available, if we use CSA by Belazzougui and Navarro [16], Theorem 1 can be restated as follows:

Theorem 3. *Given a text* T *of length* n *and a property* π*, a C-PST can be maintained in* $nH_k(1 + o(1)) + O(n)$ *bits such that property matching queries can be answered in* $O(|P| + (1 + occ_\pi)\log n)$ *time, where* P *is the online query pattern, and* H_k *is the* k*th order empirical entropy of* T*.*

Proof. The space–time bounds of CSA by Belazzougui and Navarro [16] are as follows: $|CSA| = nH_k(1 + o(1)) + O(n) + O(n \log n/s)$ bits, $t(|P|) = O(|P|)$ and $t_{SA} = s$. The above theorem can be obtained by choosing $s = \log n$ in CSA and substituting the bounds in Theorem 1. □

3.2. Construction

So far we have discussed the components of proposed index C-PST and how it can be used for property matching. In this subsection we talk about the construction of C-PST and space–time requirements for the same.

Theorem 4. *C-PST can be constructed in linear space and time.*

Proof. Initially we construct the suffix array of T, where SA and SA^{-1} operations can be performed in constant time. Suffix array construction is a well studied problem and there are many standard algorithms with linear space and time requirements [28]. Remaining components of the C-PST can be constructed as follows:

- $end(i)$ can be computed in linear time for $i = 1 \ldots n$ using following algorithm. *IF* statement in the algorithm can be evaluated in constant time by maintaining intervals (s, f) in π in the sorted order of their starting points i.e. s. This sorting can be performed in $O(n)$, if $\pi < n/\log n$, else we can use radix sort. As $end(i)$ is computed in sequential manner for $i = 1 \ldots n$, we maintain a pointer to the list of intervals sorted by s, which allows the condition in the *IF* statement to be verified in constant time.

```
end ← 0
for i = 1 to n do
  if i = s_k and f_k > end, for some interval (s_k, f_k) ∈ π then
    end ← f_k
  end if
  end(i) ← end
end for
```

- The bit vector B can be constructed in linear time by scanning $end(i)$ once, and the space and time required for constructing the *rank–select* structure on top of B is also linear [23].
- Array $A[i] = length(SA[i])$ can also be constructed in linear time as SA values can be computed in constant time. Further the construction time for RMQ over A is also linear [19].

After building these auxiliary structures, we can replace the suffix array by compressed suffix array, which also can be constructed in linear space and time [29]. □

4. Handling dynamic properties

In this section, we show a dynamic version of C-PST, which can handle insertion and deletion of intervals from property π. In dynamic C-PST, RMQ and bit vector components are simply replaced by their dynamic counterparts which are described in Section 2. In addition, dynamic C-PST also maintains all the intervals (s, f) in text property π, in the form of a binary search tree (BST) of $|\pi|$ nodes (of size $O(|\pi| \log n)$ bits). Ordering of the intervals in BST is based on their starting points i.e. s. Intervals can be searched (based on starting point), added or deleted from this BST in $O(\log |\pi|) = O(\log n)$ time.[1]

Lemma 9. *The dynamic version of C-PST occupies* $|CSA| + O(n + |\pi| \log n)$ *bits space.*

Proof. The index space for dynamic C-PST can be bounded as follows: $|CSA|$ bits CSA, $O(n)$ bits dynamic RMQ, $2n + o(n)$ bits dynamic bit vector B and $O(|\pi| \log n)$ bits BST of all intervals in π, thus total $|CSA| + O(n + |\pi| \log n)$ bits. □

Lemma 10. *Property matching can be performed in* $O(t(|P|) + (1 + occ_\pi)(t_{SA} + \log n / \log\log n) \log n)$ *time using dynamic C-PST.*

Proof. As described in Lemma 8, we begin by matching the pattern P using CSA in $O(t(|P|))$ time and obtain the suffix range and then apply at most $(2\,occ_\pi + 1)$ recursive range maximum queries to retrieve all occ_π answers. However in this case, dynamic RMQ described in Lemma 3 needs $O((1 + occ_\pi) t_{access} \log n) = O((1 + occ_\pi)(t_{SA} + \log n / \log\log n) \log n)$ time for retrieving desired occ_π occurrences. □

The following observation simplifies our task of supporting insertion and deletion of intervals from property π.

Observation 1. By the insertion or deletion of an interval (s, f) in π, we need to update $end(i)$ only for $s \leqslant i \leqslant f$.

Proof. From Definition 4, $end(i)$ depends only on those intervals which start before i. Hence, insertion or deletion of an interval (s, f) will not affect $end(i)$ for $i < s$. But for $i > f$, this may change only for those values with $end(i) < i$. Note that for those positions $(length(i) = end(i) + 1 - i \leqslant 0)$ that cannot match with any pattern and this inequality remains the same,

[1] $\pi \leqslant n(n-1)/2$.

even if we do not update (as per Definition 4) *end*. In other words, this strategy that does not follow Definition 4 strictly for i with $end(i) < i$ will not affect the correctness or performance of our algorithm.[2] Therefore we need to update $end(i)$ only for $s \leqslant i \leqslant f$. □

Lemma 11. *Given a text T of length n and a property π, a dynamic C-PST can be maintained such that the update (insert or delete) of an interval (s, f) can be performed in $O((f-s)(\log n + t_{SA}))$ time.*

Proof. Given an update to the property π, we first insert or delete the query interval (s, f) in BST and then use the following algorithm to recompute $end(i)$ for $s \leqslant i \leqslant f$.

```
end ← end(s − 1)
for i = s to f do
  if i = s_k and f_k > end, for some interval in π then
    end ← f_k
  end if
  end(i) ← end
end for
```

In each iteration, the search in BST takes $O(\log|\pi|) = O(\log n)$ time. Now t_{access} for the dynamic RMQ structure is $O(t_{SA} + \log n/\log\log n)$ and updating a bit vector entry takes $O(\log n/\log\log n)$ time and that of dynamic RMQ structure is $O(\log n)$. The number of updates in B (along with the rank–select structures) and in RMQ is also bounded by $O(f-s)$. Therefore the update time is $O((f-s)(\log\pi + t_{SA} + \log n/\log\log n + \log n)) = O((f-s)(t_{SA} + \log n))$. □

Lemmas 9, 10 and 11 collectively prove Theorem 2. Again by using CSA proposed by Belazzougui and Navarro [16] Theorem 2 can be restated as follows:

Theorem 5. *Given a text T of length n and a property π, a dynamic C-PST can be maintained in $nH_k(1+o(1)) + O(n + |\pi|\log n)$ bits space such that property matching queries can be answered in $O(|P| + (1+occ_\pi)\log^2 n/\log\log n)$ time and update (insert or delete) of an interval (s, f) can be performed in $O((f-s)\log n)$ time, where H_k is the kth order empirical entropy of T.*

5. Concluding remarks

In this paper, we have proposed space efficient indexes for property matching problem. We believe that our result for the static case is the best possible result (though any improvement in CSA will improve our index). However for the dynamic case, the update time is dependent on the interval size. It is an open question, whether we can design an index for dynamic property matching, which can handle updates efficiently (e.g. *poly*$\log(n)$ time) instead of update time being proportional to the interval size.

References

[1] U. Manber, E.W. Myers, Suffix arrays: A new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[2] R. Grossi, J.S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, SIAM J. Comput. 35 (2) (2005) 378–407, a preliminary version appears in STOC'00.
[3] P. Ferragina, G. Manzini, Indexing compressed text, J. ACM 52 (4) (2005) 552–581.
[4] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (1) (2007).
[5] J. Jurka, Human repetitive elements, in: R.A. Meyers (Ed.), Molecular Biology and Biotechnology, 1995, pp. 438–441.
[6] J. Jurka, Origin and evolution of Alu repetitive elements, in: R. Maraia (Ed.), The Impact of Short Interspersed Elements (SINEs) on the Host Genome, 1995, pp. 25–41.
[7] A. Amir, E. Chencinski, C.S. Iliopoulos, T. Kopelowitz, H. Zhang, Property matching and weighted matching, Theor. Comput. Sci. 395 (2–3) (2008) 298–310.
[8] C.S. Iliopoulos, C. Makris, Y. Panagis, K. Perdikuri, E. Theodoridis, A. Tsakalidis, The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications, Fundam. Inform. 71 (2006) 259–277.
[9] H. Zhang, Q. Guo, C.S. Iliopoulos, An algorithmic framework for motif discovery problems in weighted sequences, in: International Conference on Algorithms and Complexity, 2010, pp. 335–346.
[10] C.S. Iliopoulos, M.S. Rahman, Faster index for property matching, Inf. Process. Lett. 105 (6) (2008) 218–223.
[11] M.T. Juan, J.J. Liu, Y.L. Wang, Errata for "faster index for property matching", Inf. Process. Lett. 109 (18) (2009) 1027–1029.
[12] T. Kopelowitz, The property suffix tree with dynamic properties, in: Proceedings of Symposium on Combinatorial Pattern Matching, 2010, pp. 63–75.
[13] H. Zhao, S. Lu, Compressed index for property matching, in: Proceedings of Data Compression Conference, 2011, pp. 133–142.
[14] P. Weiner, Linear pattern matching algorithms, in: Proceedings of Symposium on Switching and Automata Theory, 1973, pp. 1–11.
[15] K. Sadakane, Compressed suffix trees with full functionality, Theory Comput. Syst. 41 (4) (2007) 589–607.

[2] If $end(i) < i$, then $end(i)$ returns some arbitrary value $j < i$.

[16] D. Belazzougui, G. Navarro, Alphabet-independent compressed text indexing, in: Proceedings of European Symposium on Algorithms, 2011, pp. 748–759.
[17] B. Chazelle, A functional approach to data structures and its use in multidimensional searching, SIAM J. Comput. 17 (3) (1988) 427–462.
[18] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: Proceedings of Latin American Symposium on Theoretical Informatics, 2000, pp. 88–94.
[19] J. Fischer, V. Heun, Space-efficient preprocessing schemes for range minimum queries on static arrays, SIAM J. Comput. 40 (2) (2011) 465–492.
[20] G. Jacobson, Succinct static data structures, PhD thesis, Carnegie Mellon University, 1989.
[21] D.R. Clark, J.I. Munro, Efficient suffix trees on secondary storage (extended abstract), in: Proceedings of Symposium on Discrete Algorithms, 1996, pp. 383–391.
[22] R. Pagh, Low redundancy in static dictionaries with o(1) worst case lookup time, in: Proceedings of International Colloquium on Automata, Languages and Programming, 1999, pp. 595–604.
[23] R. Raman, V. Raman, S.R. Satti, Succinct indexable dictionaries with applications to encoding *k*-ary trees, prefix sums and multisets, ACM Trans. Algorithms 3 (4) (2007).
[24] R. Raman, V. Raman, S.S. Rao, Succinct dynamic data structures, in: Proceedings of Workshop on Algorithms and Data Structures, 2001, pp. 426–437.
[25] W.-K. Hon, K. Sadakane, W.-K. Sung, Succinct data structures for searchable partial sums with optimal worst-case performance, Theor. Comput. Sci. 412 (39) (2011) 5176–5186.
[26] V. Mäkinen, G. Navarro, Dynamic entropy-compressed sequences and full-text indexes, ACM Trans. Algorithms 4 (3) (2008).
[27] A. Gupta, W.-K. Hon, R. Shah, J.S. Vitter, A framework for dynamizing succinct data structures, in: Proceedings of International Colloquium on Automata, Languages and Programming, 2007, pp. 521–532.
[28] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, J. ACM 53 (6) (2006) 918–936.
[29] W.-K. Hon, K. Sadakane, W.-K. Sung, Breaking a time-and-space barrier in constructing full-text indices, in: Proceedings of Symposium on Foundations of Computer Science, 2003, pp. 251–260.